SIENAcollege
Computer Science

Computer Science 340
Programming Languages
Siena College
Fall 2019

# Topic Notes: Abstraction and Encapsulation

*Abstraction* and *encapsulation* are fundamental concepts in nearly all modern programming languages.

*Abstract data types* are user-defined (or perhaps language API-defined) data types that

- hide implementation details from users of the types
- provide a limited and restricted set of operations to modify and query the state of instances of the type
- are normally defined by a single syntactic unit (*e.g.*, a *class*)
- other parts of the program can then declare, allocate, and use instances of this new data type

Data abstraction improves reliability, writability, and modifiability, hence is supported in nearly every modern and not-that-modern programming language.

- hidden data representation of the ADT makes it easier for users of the ADT – they don't need to know or care about the internal details
- meanwhile, ADT authors can change internal representation without affecting code using it (assuming its public interface remains)
- natural way to organize code for separate implementation, separate compilation
- reusability!

We are all familiar with examples of this, in *e.g.*, Java:

**See Example:**
`/home/cs340/examples/javageneric`

A C++ example:

**See Example:**
`/home/cs340/examples/cppvector`

Note that there are different levels of *information hiding* possible: `public`, `private`, `protected` data members and/or methods.

Most ADTs come equipped with special methods called *constructors*, and in some cases *destructors*, which are called implicitly on creation, deletion, of instances of the ADT, respectively.

In languages without explicit support for encapsulation, such as C, we can still do separate compilation, but there is no guarantee that users of the ADTs will not modify the state of the structure without using provided functions.

**See Example:**
`/home/cs340/examples/ratios`

---

# Language Requirements

A programming language that supports ADTs must have:

- a syntactic unit to support encapsulation, *e.g.*, class or package
- a mechanism to separate information such as method signatures available to callers while hiding the implementation details

In the language design phase, some decisions need to be made:

- what does the ADT container look like?
- are type parameters (*i.e.*, generics) allowed?
- do we separate the interface definition from the implementation details (*e.g.*, as in C++ with separate header and implementation files for a class)?

The text describes examples of *encapsulation mechanisms* in some detail, we will look at C++. Refer to the text for more about the mechanisms in C# and Ruby.

- C++ uses a `class` as the encapsulation mechanism
- C++ classes look a lot like C `structs`
- there is only one copy of the code of the class functions shared among all instances
- each instance gets a copy of class data members (*i.e.*, instance variables)
- C++ classes can be allocated statically (as globals or `static` members), dynamically on the stack (declare as local variables), or dynamically on the heap (with `new`)
- we saw the protection levels for information hiding previously: `public`, `protected`, `private`
- constructors in C++ are functions, whose name must match the class name, intended to initialize instance variables, allocate any dynamic structures
- constructors are called implicitly when a class is instantiated but can also be called explicitly like any other function

- destructors are needed as C++ memory is explicitly deallocated so any dynamic memory associated with the instance of a class must be returned to the system with `delete`
- destructors are named with the class name also, but preceded by ~
- destructors also called implicitly but can be called explicitly
- C++ separates the interface (in a header file) from the implementation (in a code file)
- in addition to standard protection levels, C++ allows `friend` functions or classes that are permitted access to private members

Many of us are most familiar with Java – it borrows many ideas from C++, adding a new scoping mechanism called *package scope* instead of friends.

---

## Parameterized ADTs

When type parameters are allowed, giving rise to generic structures, additional issues must be considered. The idea is that we can write one ADT that can store any (or almost any) type of elements, while maintaining static typing (ensuring only the proper type of elements can ever be added, removed, etc. from our ADT).

Again, the text has some examples, including Ada and C#. In C++, generics are implemented as templated classes. The syntax and idea is similar to that in Java.

- in C++, the type can be anything, including primitive types
- in Java, the type has to be an `Object` type, necessitating the container classes (`Integer`, `Double`, *etc.*)
- Java's introduction in version 5.0 of these eliminated the need for casts when retrieving elements (and when combined with *autoboxing* and *autounboxing* allows nearly seamless use of primitive types)
- side note: when these were introduced, Sun (the company that developed the language) wanted to make sure programs using generic types could run on prior versions of the Java Virtual Machine (JVM), so type parameters are used at compile time but that information is erased before run time

  **On the web:** Type Erasure in the Oracle Java Tutorials at
  `https://docs.oracle.com/javase/tutorial/java/generics/erasure.html`

**See Example:**
`/home/cs340/examples/javaerasure`

---

# Namespace Encapsulations

Large programs bring together code that defines a large number of global names (*e.g.*, functions, global variables), raising the chances of collisions among those names.

In a language like C, programmers can agree on some conventions to reduce the chances of collisions. For example, all global names defined by the Message Passing Interface (MPI) for C programs start with the prefix `MPI_`.

Other languages provide more explicit support for *namespace encapsulation*:

- C++ namespaces: a library can be implemented with all of its names within a namespace, and that namespace would be used to access those names externally

  ```
  use namespace std;
  ```

  allows all of the names in the `std` namespace to be used by the remainder of the file without the prefix qualifier `std::`

- Java package system allows additional access (to `protected` members) among package classes

  ```
  import java.util.Scanner;
  ```

  allows the name `Scanner` to be used from package `java.util` without specifying the fully qualified name `java.util.Scanner` within the code